Matt Williamson
CS410
November 14, 2020

AWS Lambda as a Distributed Information Retrieval Service Host

An attempt and assessment of suitability for prototyping or research activities

As noted in the course lectures, one significant advancement that paved the way for the modern World Wide Web and search engines such as Google was the MapReduce mechanism which allowed for processing of indexes and other tasks in a way that was massively parallel and highly scalable. A side effect of the development of these techniques is that similar mechanisms have been generalized and made available as a commodity product through cloud service providers. One such service (among several) is the Lambda function capability provided by Amazon Web Services (AWS); this product facilitates the development of small, pipeline-like functions that are activated and executed on-demand, and can scale quickly and inexpensively to create large, highly concurrent data processing systems—for instance it has already been shown how this service can be used to implement a MapReduce system.[1]

For this review, it is particularly relevant that the AWS Lambda framework allows the creation of functions with Python (currently versions 3.6-3.8) including the import of arbitrary libraries as dependencies, and also provides a build toolchain for properly compiling libraries that require compiled C code—such as MeTA/metapy—so that they will execute correctly in the target cloud environment.[2] What follows are key points and observations resulting from attempting to create a simple text retrieval REST service using the metapy libraries within an AWS Lambda function.

---

[1] https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/

[2] https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-using-build.html

The produced proof of concept code is available in this review's GitHub repository,[3] including a few historical earlier versions. The text retrieval service allows for execution of simple text queries against an index (the Faculty Search dataset was used for this demo) and basic pagination of results. While largely beyond the scope of this review, and fairly well documented elsewhere,[4] the toolset used to build the demo was Amazon Web Service's Serverless Application Model command line interface (AWS SAM CLI). The SAM CLI allows for building a Lambda in a Docker container that emulates the target environment as mentioned earlier.

The initial attempt[5] at building a basic working Lambda function was straightforward, simple, and successful. The necessary python code[6] was easily written and mostly devoid of environment-specific complexity; the Lambda infrastructure provides a dictionary with the request parameters received from an AWS API Gateway, so most of the concerns of handling HTTP and REST services are abstracted away—the only complexity was self-imposed: that of trying to write a single function that would handle both "GET' and "POST" requests. The SAM CLI facilities for building and testing the function locally worked as expected and produced correct results. The MeTA/metapy libraries also compiled properly in the build environment (Python 3.7 was selected) and executed properly in the cloud environment as documented.

However, when this minimal version of the Lambda function was deployed to the cloud environment, the hidden complexities inherent in such a modern system began to show themselves. While the compiled MeTA library was fully functional, specifics of the way it was used caused an incompatibility that was not apparent when testing the function in the local environment. Attempting to read the prepared inverted disk index resulted in the following error:

```
[ERROR] RuntimeError: error obtaining file descriptor for ./data/idx/inv/
metadata.index
```

A reasonable expectation for the cause of this would be a path difference between local emulation and production environments, but this proved not to be the case: the data was found

---

[3] https://github.com/SphtKr/CS410_TechReview_AWSLambda

[4] https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-getting-started.html

[5] https://github.com/CS410Fall2020/tech_review/tree/3cf005a39766b4660211f30cfb0974f000e68fcf

[6] https://github.com/CS410Fall2020/tech_review/blob/3cf005a39766b4660211f30cfb0974f000e68fcf/metapy-ir-demo/app.py

where expected and this error occurred *after* reading the config.toml file in the same directory, which was found and read successfully.

Further research pointed out a likely cause. One user experiencing a similar error from MeTA showed their exception as occurring in a module called `mmap_file`[7]—`mmap` is a POSIX system-level function that reads and maps whole files into memory efficiently using OS-level capabilities. A record was also found of a user finding difficulty using mmap system calls on Amazon's related EC2 (Elastic Compute) infrastructure.[8] Furthermore, while some aspects of AWS Lambda's internal workings are proprietary or undocumented, it seems that its runtime leverages Linux containerization techniques[9] (similar to Docker)—which perform sophisticated manipulations at the OS level to present virtual resources to the process, to include the filesystem.[10] Initially, for the purposes of the proof of concept, the inverted disk image data itself was being "burned in" with the Lambda code as part of the deployment package, which is presented to the Lambda process as a read-only filesystem layer—so it seemed likely that a combination of these factors was causing the read of the index data (likely via `mmap`) to fail.

In a real-world scenario, such index data would not be stored as part of the Lambda itself but would reside elsewhere, so this problem did not accurately represent a barrier to practical use of Lambda functions. However, it does provide an example of where unexpected complexity may be introduced when relying on complex proprietary infrastructures to support non-trivial workloads. To move forward, and more accurately assess the AWS Lambda stack, it would be necessary to move the index data to a more realistic location. Unfortunately this would add significant complexity to the implementation as will be shown.

There are essentially two options within AWS to provide persistent data storage to a Lambda function: S3 (Simple Storage Service) "buckets" and EFS (Elastic File System) volumes. Most AWS "stacks" tend to rely on S3 buckets for "blob" (Binary Large OBject) storage, but in this case it was likely an unsuitable option: MeTA expects to access index data files via traditional OS-level filesystem mechanisms, and S3 only provides its own specialized API to access data

---

[7] https://forum.meta-toolkit.org/t/where-to-get-20newsgroups-dataset-that-contains-20newsgroups-dat/453

[8] https://forums.aws.amazon.com/thread.jspa?threadID=25165

[9] https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/

[10] https://medium.com/@BeNitinAgarwal/docker-containers-filesystem-demystified-b6ed8112a04a

objects directly from code; so if S3 was used to store the index data it would be necessary to copy the files to the local temporary filesystem on each invocation, likely negating the efficiency of the platform and possibly encountering the same problem with `mmap` calls. EFS on the other hand is an implementation of traditional UNIX NFS (Network File System), so seemed a far better (and more likely successful) option.

Ultimately, moving the index data to an EFS volume resolved the issue as expected and produced a fully functioning text retrieval service with a realistic implementation design. However, this was not without difficulty. Prior to this modification, the entire capability was implemented with only two AWS components in the stack: Lambda and API Gateway, and these two were integrated automatically via implicit resource allocations created and managed by the SAM CLI.[11] Adding EFS to the stack introduced significant complexity in two primary areas:

1. Role-Based Access Control: AWS is designed with a robust security infrastructure, and this is desirable. However, up until this point, permissions and security groups necessary to allow the API Gateway to talk to the Lambda and vice-versa were all handled implicitly by SAM CLI. To add an EFS volume, at least some portion of that access control would have to be managed manually—including to facilitate initial population of the data, which will be described briefly later.
2. Networking Configuration: Since EFS and NFS are fundamentally network based services, this introduced the need to configure the Lambda function to access the network. This means primarily the VPC (Virtual Private Cloud) network in which the EFS volume resided, but since the networking stack in AWS is designed for multi-site load balancing and fault tolerance, the network environment is complex and requires (for example) configuration for multiple subnets even within a single AZ (Availability Zone).

Furthermore, even after the necessary configuration was determined, implemented, and proven, there was the question of how to load the index data to the EFS volume in the first place. AWS's target audience for migrating data into their cloud environment is clearly large enterprises—hence, the documentation and toolset for this topic is complex and robust, and involves configuring and running an "Appliance" in a virtual machine within the local network and

[11] https://github.com/awslabs/serverless-application-model/blob/master/docs/internals/generated_resources.rst#api

migrating data via the virtual appliance[12] (which incidentally ships configured to use 4 CPU cores and 16GB of RAM on the host system). A brief attempt to use this toolset to load 14MB of data into EFS was quickly abandoned. In the long run, the simplest solution was to deploy a dedicated EC2 instance within the same VPC, mount the EFS volume to the EC2 instance, and then transfer the index data to the EC2 instance via SFTP and then into the mounted EFS volume—which is not to say that this was "simple", as it required establishing the networking and security rules necessary to make this all possible for the EC2 instance as was briefly discussed above for the Lambda function.

Similar complexity was encountered when adding the last planned additional facet to make the service more realistic: Swagger/OpenAPI service documentation. While supported by the API Gateway and directly by the SAM CLI, configuration again developed into something far beyond trivial. The seemingly simple addition of both "GET" and "POST" handlers caused unexpected complexity with the security infrastructure: when the API Gateway is configured via an OpenAPI specification, it expects to pass all requests to the backing Lambda via "POST"[13], and while the seemingly simple solution would be to point the Gateway's "GET" handler to the existing "POST" Lambda handler, *the implicitly managed permissions only allow it to reach the "GET" handler*—so, paradoxically, the *simplest* solution was instead to make the "GET" Lambda handler code also handle "POST" requests!

Nevertheless, all of these complexities were eventually surmountable. The final product[14] is functional and even potentially useful, and does suggest that the use of AWS Lambda for several similar scenarios is not only viable but may be advantageous.

In sum, the major potential impediment to using AWS Lambda for this or similar IR requirements is not any inherent limitation of capability, and in fact it seems highly capable. Rather, it is the ancillary complexity and relatively novel mechanisms involved in the configuration of a working system for non-trivial applications. Wrangling AWS as a whole requires a level of knowledge comparable to that required for a traditional server operating

---

[12] https://aws.amazon.com/datasync/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc

[13] https://aws.amazon.com/premiumsupport/knowledge-center/api-gateway-lambda-template-invoke-error/

[14] https://app.swaggerhub.com/apis-docs/MetaPy-IR-Demo/metapy_information_retrieval_demo/0.1.0#/default/searchGet

system, and while the principles are similar to those underlying traditional OS's, AWS does not have as large or old a base of experienced administrators.

If that risk can be accepted, there are definitely potential advantages. Some motivations for the use of Lambda functions have been discussed, particularly on-demand scalability—that is, if no one is accessing your service, you are using zero resources, and if 1,000 people access your service at the same time, you can suddenly effectively run 1,000 instances of your service at once. Crucially, this *can* have significant impacts on the costs of running such a service: if you ran a service such as the one described in this review using traditional cloud-based virtual machine infrastructure, you would be charged for all of the execution time that your virtual server was sitting idle serving no users (or, you would have to know exactly when to shut down the server when there was no demand to avoid being charged). Likewise, if you provisioned a virtual server capable of supporting 100 concurrent users, it would not effectively support a burst of 1,000 users (or would also require more complex automatic scaling configuration). This could be an important capability in many research activities involving human input (e.g. IR performance validation) where the provider does not have control over when subjects or contributors may be using the system. It is also of clear benefit in many production and even commercial scenarios—this approach may be extremely useful and cost-effective especially in early iterations of a commercial venture in sectors such as vertical search or fielding of novel retrieval algorithms or techniques. Other unique capabilities within S3 may have similar benefits for certain workloads and when combined with Lambdas—for instance, while EFS was needed for this application, S3 is designed for efficient and cost-effective use of very large datasets, and with sharding techniques or a library that directly supports S3 it is very likely possible to extract similar efficiencies and savings.

As a small data point, it is worth observing that the total cost incurred from all AWS services in the production of the proof of concept capability described herein amounted to $0.42—and that figure is attributable entirely to the small amount of time needed to run the EC2 instance *to import the initial index data into the EFS volume*. The AWS free tier includes 1,000,000 Lambda executions and 400,000 seconds of Lambda compute time per month, and since all development of the Lambda function itself was done locally, the whole use of the Lambda capability had zero cost incurred. With traditional virtual machine IaaS (Infrastructure as a

Service) system design, likely at least an order of magnitude more cost would be incurred merely for the compute time necessary to install, configure, and test the system within an EC2 instance.